High Performance Computing - HW 6

Anthony Maylath

May 13, 2019

The Courant Institute for Mathematical Sciences, New York University

1 Question 0

We have completed an MPI implementation of the Kmeans and EM algorithms. At this point, Kmeans is pretty optimized while we are still improving the EM implementation. We ran scalability and timing tests on Kmeans and found descent performance. Our plan is to polish our EM implementation using similar HPC techniques and compare performance and results between the two. We have not yet started compiling the report or presentation slides.

2 Question 1

My implementation can be found *jacobi_mpi.cpp*. The implementation takes two command line arguments: the first sets the dimension of the problem, N, and the second specifies the max number of Jacobi iterations. My implementation uses row major ordering for the solution lu. I send a different communication depending on where the process falls on the grid. For instance, if the process computes the solution on the top row, then don't comunicate ghost values for the top. I filter out top row comunication by checking if $mpirank >= \sqrt{p}$ where p is the number of processes. I apply similar checks on all four edges of ghost values.

Figures 1 and 2 demonstrate the weak scalling of the implementation for 64 nodes with 4 cores per node and 16 nodes with 16 cores per node respectively. Each line represents a fixed number of Jacobi iterations. The base size of the problem is $N = 100$ and is scalled up proportional to the number of processes so that each process gets $lN = 100$ locally. The secondary axis corresponds to 1,000 and 100 iterations. For 100 iterations, it seems like we lose some performance for larger problem sizes, while the performance time seems more consistant across problem sizes for more iterations.

Figure 3 shows strong scaling log timings with $lN = 25,600$. The results were obtained with ? nodes and 16 cores on Prince. To get the expected time, I take the serial time and divide it by the number of processes. The strong scaling appears to be quite good with the actual timing falling well below expectation. Of course, this could be due to transiant artifacts that made the serial timing unusually long.

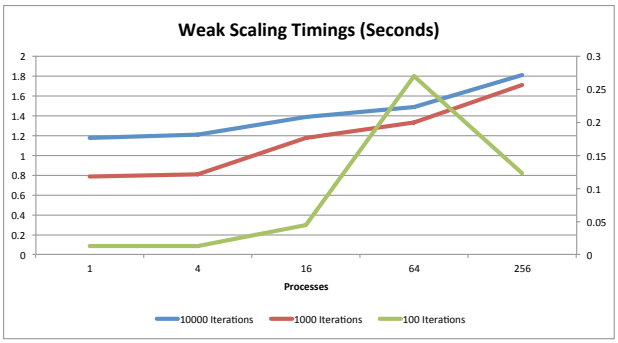

Figure 1: Weak Scaling 64 Nodes, 4 Cores Per Node

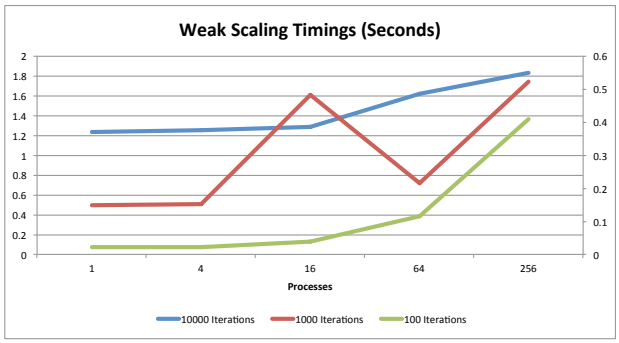

Figure 2: Weak Scaling 16 Nodes, 16 Cores Per Node

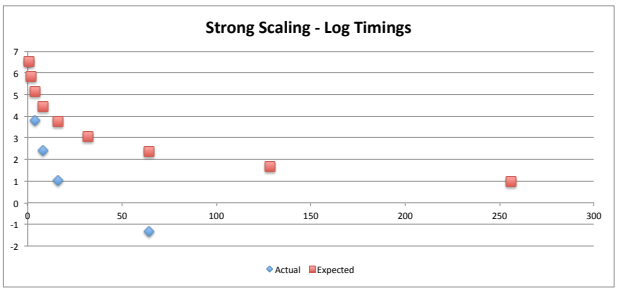

Figure 3: Strong Scaling ? Nodes, 16 Cores Per Node

2.1 Voluntary Bonus Question: Non-Blocking Jacobi

File jacobi_noblock.cpp contains a non-blocking version of a distributed Jacobi solver. I define three new functions. localJacobi() computes the next Jacobi iteration, just like the blocking version. innerJacobi() only computes the Jacobi step for points not adjacent to a ghost point. outerJacobi only computes the Jacobi step for points adjacent to ghost points.

Each time localJacobi() is called, I send the top and bottom ghost values and compute innerJacobi(), while the values send. Then, I send the left and right ghost points and wait for all the recieves to complete before computing outerJacobi(). A new array, *lwait*, holds the results of inner and outerJacobi() so all three steps are persisted.

I ran the blocking and non-blocking over 16 nodes with 16 tasks per node for several different max iterations and dimension size. The results can be seen in table 1. The *Improvement* column shows the timing of the blocking implementation minus the timing of the non-blocking implementation. Overall, the non-blocking version seems slightly slower than the blocking implentation most of the time. There is one instance where blocking does significantly better with 100 iterations, 256 processes and 256 points. Also notice the improvement tends to be larger for fewer iterations.

Processes	N	Iterations	Time Blocking	Improvement
1	100	100	0.015849	-0.003424
4	400	100	0.022168	0.00311
16	1600	100	0.016487	-0.014867
64	6400	100	0.082373	-0.042053
256	25600	100	0.373123	0.257462
1	100	1000	0.15263	-0.018995
4	400	1000	0.186669	-0.005034
16	1600	1000	0.183917	0.014694
64	6400	1000	0.209188	-0.206038
256	25600	1000	0.271212	-0.295981
1	100	10000	1.17058	-0.024951
4	400	10000	1.348892	-0.044315
16	1600	10000	1.316946	-0.040889
64	6400	10000	1.733511	0.054059
256	25600	10000	1.877382	-0.028926

Table 1: Non-Blocking Improvement on Prince

3 Question 2

sample_sort.cpp contains my implementation of mpi sample sort. I start by assigning each process a list of integers of size $lN = N/p$ where N is specified as the first command line argument. The local lists are initialized with the $rand()$ function in the standard library. The spliter candidates are selected to be the first $p-1$ elements of each local list, where p is the total number of processes. Since the numbers are randomly initialized, these first elements will have random order. To perform local sorting, I use quicksort. The implementation is located in header file, $hw6helper$, and is a modification of a quicksort algorithm from stack overflow. The sorted output is stored in files named $output(rank).txt$ where rank is the local mpi rank. Each file has elements in ascending order and files with larger ranks contain larger elements.

I ran my timings on Prince with 16 nodes, 16 tasks per node, and 16GB of memory. Figures 4 and 5 graph the strong and weak scaling respectively. Each symbol in figure 4 represents a list size. We see larger list sizes give better strong scaling. Also, the performance seems to go down if we use too many processes. The optimal processes seems to be 16 and 50 for list size 1 million and 10 million respectively. Perhaps we need a larger list size to see optimal performance with over 100 processes. Figure 5 keeps the local list size as 10,000. The global size is increased with the number of processes. The run time goes up as we increase the problem size, except when we move from 4 to 10 processes when it goes down. The trend line indicates a slope of 0.0148 which implies that about 0.02 seconds are added to the run time when the list size is increased by 10,000 and the number of processes are increased by 1. The R^2 of the trendline is quite high at 0.998 which suggests the weak scaling really is linear; however, we probabaly need more datapoints to make the result statistically significant.

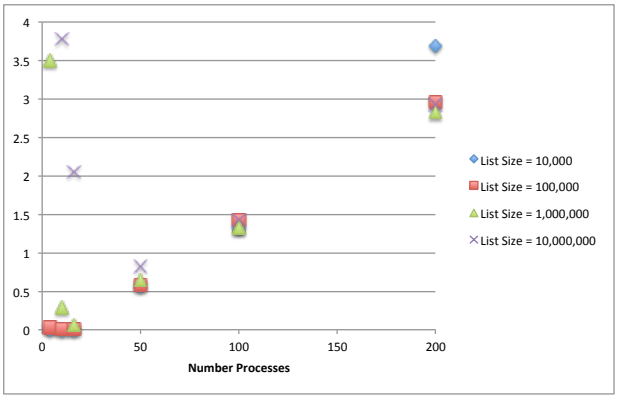

Figure 4: Sample Sort: Strong Scaling Timings

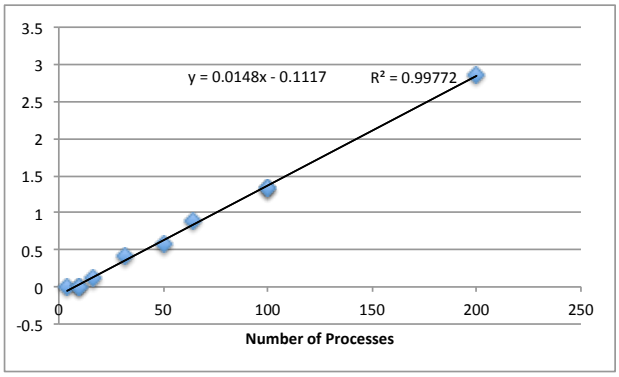

Figure 5: Weak Scaling for 10,000 Points per Process - Timings

4 Question 3: Extra Credit

I opted to parallize the one dimensional multigrid method in openmp. The my implementation is an alteration of the code available on git for lecture13. Note that I use g++-8 to compile my code. The implementation now requires 4 command line parameters with the fourth parameter being the number of omp threads. I added parallel for statements to the set_zero, compute_residual, and jacobi functions. In addition, I saw speed up by adding parallel for statements to the coarsen and refine_and_add functions despite an exponentially decreasing amount of work with each v-step. These functions also have parallel for pragmas in my implementation. Instead of performing a memcpy with each jacobi step, I simply switch the u and $unew$ pointers. This speeds things up, but requires the $ssteps$ parameter to be even. Finally, I added a reduction to the function $compute\_norm$.

Figures 6 and 7 show the strong and weak scaling for omp threads $= 1, ..., 20$ and $1, ..., 11$ respectively on Prince with 1 node, 20 cores, and 60BG of memory. For strong scaling, I always use 700 million points. For weak scaling I start with 100 million points and add another 100 million points for each additional omp thread. For strong scaling, the optimal number of omp threads for 700 million points seems to be between 10 and 15. For weak scaling, a line fit estimates that each omp thread adds about 5 seconds of run time. The weak scaling isn't great but its much better than running serially as the base case serial code takes about 9 seconds for 100 million points. If we scale this up linearly, that implies the serial code should take around 90 seconds for 1 billion points. However, we see better performance on the parallel implemenation with around 36 seconds on 1 billion points and 10 threads.

For a comparison, I also ran scalings on crakle2 which can be seen in figures 8 and 9 respectively. For strong scaling, the optimal number of threads comes in at 4 with a timing of 38 seconds. On Prince, the best time was slightly lower at 33 seconds with 11 threads. The weak scaling is also a bit worse than Prince with an estimated slope of approximately 6. Prince only performs slightly better with much more hardware due to the constraint of shared memory.

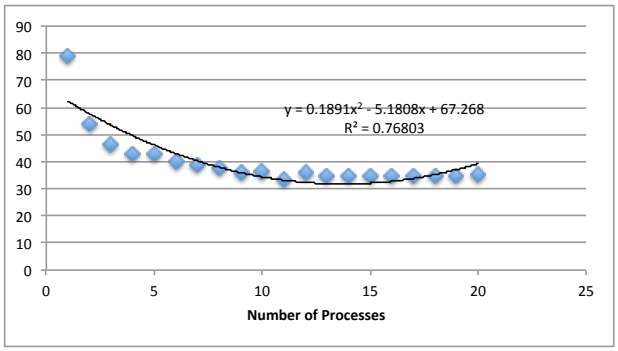

Figure 6: Prince - Strong Scaling Timings

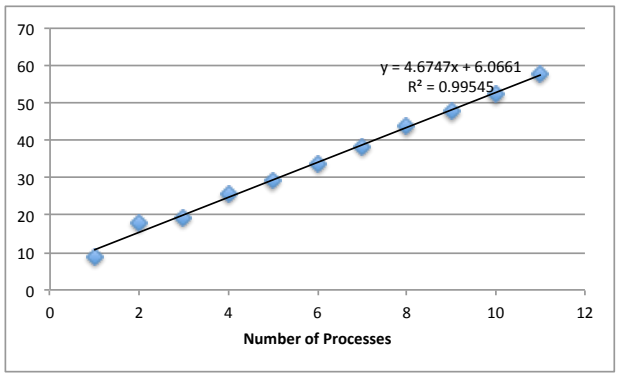

Figure 7: Prince - Weak Scaling Timings

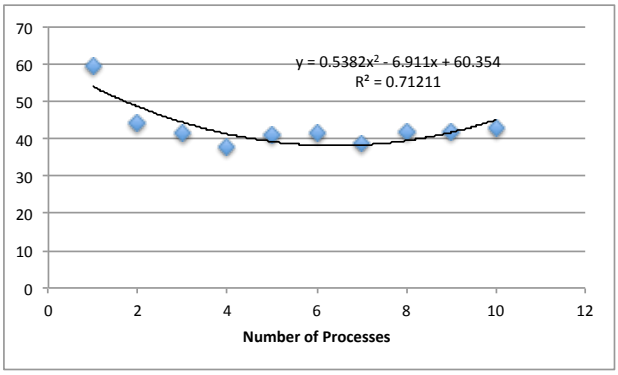

Figure 8: crackle2 - Strong Scaling Timings

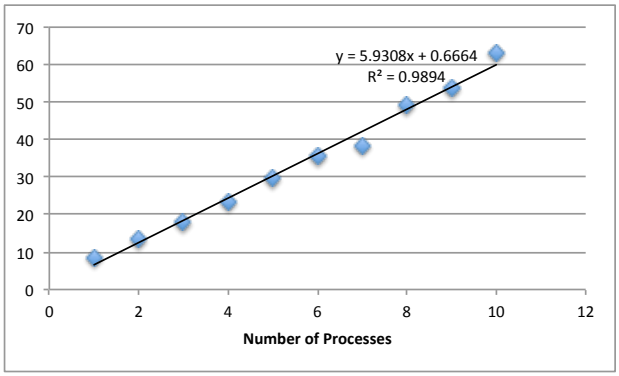

Figure 9: crackle2 - Weak Scaling Timings